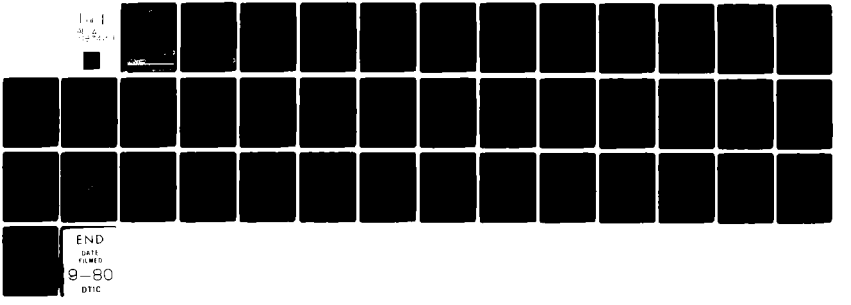
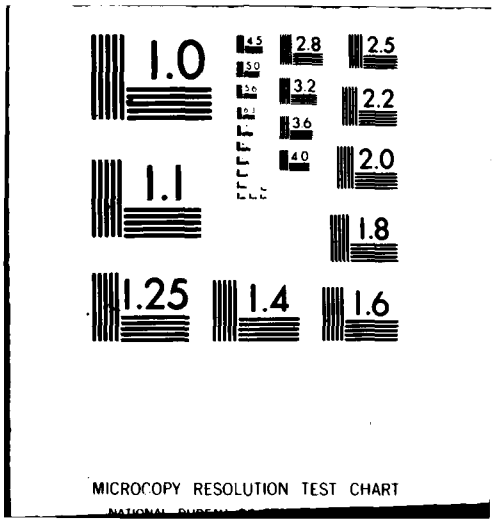
AD-A087 420 LOCKHEED MISSILES AND SPACE CO INC PALO ALTO CA PALO --ETC F/G 5/1
DATABASE MANAGEMENT IN SCIENTIFIC COMPUTING II. DATA STRUCTURES--ETC(U)
FEB 79 C A FELIPPA

UNCLASSIFIED LMSC/D673048 NL



END
DATE FILMED
9-80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART

LEVEL II

AD A087420

DTIC
ELECTE
AUG 4 1980
S D
D

DDC FILE COPY

LOCKHEED

MISSILES & SPACE COMPANY, INC • SUNNYVALE CALIFORNIA

DISTRIBUTION STATEMENT A
Approved for public release;

80 8 9 083

(14) LMSC/D673048

(6)

DATABASE MANAGEMENT IN SCIENTIFIC COMPUTING

II DATA STRUCTURES AND PROGRAM ARCHITECTURE

by

(10) CARLOS A. FELIPPA

(11) FEBRUARY 1979

(12) 40

APPLIED MECHANICS LABORATORY

ORGANIZATION 52-33

LOCKHEED PALO ALTO RESEARCH LABORATORY

LOCKHEED MISSILES & SPACE COMPANY, INC.

PALO ALTO, CALIFORNIA 94304

ABSTRACT

This second of a three-part paper on scientific database management prepares the ground for the operational description of a multilevel data management system in Part III. Following a detailed categorization of data and storage structures, principles of program architecture deemed necessary to make most efficient use of centralized data management techniques are discussed. General operational requirements of data management systems are presented, and the notion of activity levels introduced. This part closes with a brief review of the evolution of data management techniques in the context of computerized structural analysis.

CONTENTS

Paragraph — Page

SECTION 1 INTRODUCTION

SECTION 2 BASIC TERMINOLOGY

SECTION 3 PROGRAM ORGANIZATION

SECTION 4 DATA MANAGEMENT SYSTEMS

SECTION 5 ACTIVITY LEVELS

APPENDIX A DATA MANAGEMENT IN COMPUTERIZED STRUCTURAL ANALYSIS

ILLUSTRATIONS

Figure — Page

1 Logical Data Structure Hierarchy (Single Arrow Indicates A One-to-one Relationship, A Double Arrow A One-to-many Relationship) 2-6
2 Operational Components of A DMS-linked Running Program 4-4
3 High-level Data Flow Diagram of IPN Example: Interactive Preliminary Design of Large Space Structures 4-5
4 Three-dimensional Visualization of the Operation of the IPN Example 4-6
5 A Layered (Multilevel) Data Management System 5-2
6 Sample Configurations in Which the Layered Data Management System of Figure 5 Can Be Used in Support of A User Program (P) 5-3
7 Macro- and Micromanagement Illustrated in Conjunction With the Use of A Query Controller 5-5

TABLES

Table — Page

1 Terminology Pertaining to Computer Storage Facilities (Arranged Alphabetically) 2-3
2 Terminology Pertaining to Logical Data Objects (Arranged by Increasing Degree of Logical Complexity) 2-7
2A Special Types of Data Aggregates 2-8
2B Further Categorization of the Term "Database" 2-9
3 Terminology Used to Describe Logical Data Structures in Various Data Management Systems 2-10
4 Terminology Pertaining to Physical Storage Structures (Roughly Arranged by Increasing Complexity) 2-11

Table		Page
5	Terminology Pertaining to Program Architecture (Alphabetically Arranged)	3-4
A-1	Evolution of Computerized Structural Analysis	A-4

Section 1

INTRODUCTION

The first part of this paper [1] describes general features of scientific data management from a *functional* standpoint. Emphasis is placed on what the data management software is supposed to do for end users of applications programs, rather than how it does it. On the other hand, the third part is devoted to the analysis of the implementation of a specific data management system, which is offered as a case study. The description level in Part III is aimed at fairly experienced developers of advanced application programs rather than at users.

Part II is intended as a "bridge" between Parts I and III. The easy-to-follow but superficial descriptive style of Part I has to give way to more precisely stated concepts. Inasmuch as data management technology, and especially scientific database management, is still in a formative stage, pertinent terminology is far from standardized. Organizations engaged in some form of data management software development (research laboratories, software houses, university departments, engineering groups) have evolved "local dialects". Dialect coalescence is occurring at the physical description level, but wide semantic discrepancies remain at the logical description level. A substantial portion of Part II is therefore devoted to cataloguing terminology appropriate for scientific data structures. Although no claims are made as regards the advantages of the proposed nomenclature, it is hoped that correlations such as presented, for instance, in Table 3 may facilitate the reading of pertinent literature.

Once semantic questions are put behind, the role played by a data management system in support of an applications program, or a network of such programs, is described. The description requires an understanding of the type of modular architecture that is deemed necessary to make centralized data management effective. The concept of logical and physical module is introduced in this context. Following this, the function of the data management system can be precisely defined.

In the last section, the fundamental notion of activity levels is introduced. This is an important feature that distinguishes the *implementation* of scientific database management from that of current business data management systems. Although the hierarchical database model is stressed, the likely appearance of *micromanagers* and *program control systems* based on relational models is mentioned.

The material covered in this part does require a level of familiarity with the subject which exceeds that assumed in Part I. (In particular, a modest acquaintance with the educational references cited in Part I would be helpful). Nevertheless, most of the discussion is independent of the particular implementation described in Part III.

An Appendix contains a brief review of the evolution of data management techniques in the context of computerized structural analysis, a subject omitted from Part I for space reasons. It is intended to provide some historical perspective to readers familiar with the computer implementation of finite element methods.

Section 2

BASIC TERMINOLOGY

An introductory study of the subject of generalized data management is, in many respects, an exercise in semantics. This process categorizes and formalizes the description of many objects and activities with which experienced computer programmers and end users are intuitively familiar.

The terminology used for describing entities associated with data management has varied substantially from one authority to another, and even from time to time within the same organization. It is therefore appropriate to present here a summary of the nomenclature that will be needed in Parts II and III. (The definitions are more precise but sometimes less intuitive than those given in the Glossary of Part I.)

Knowledgeable readers are advised to proceed directly to Section 3 after a cursory examination of the tables and figures included in this section; these may serve as backup references while reading the ensuing material.

Data and Storage Structures

Information can be characterized as measurable or quantifiable knowledge. Information recorded in a physical medium becomes *data*. A *digital computer* is a machine capable of performing operations on data recorded in digital or numerical form. Such data is held in *storage facilities* or *memories*. A glossary of various terms commonly used in connection with storage facilities is arranged alphabetically in Table 1.

The data space of a computer is not an amorphous collection of bits and bytes, as it might appear upon looking at a typical memory dump. There is a logical structure lurking there, i.e., some method in that madness. A set of logically interrelated data objects identified by a name is called a *data structure*. Well-known examples are scalars, arrays, linked lists and tables. Data structures of primary importance in scientific computation are formally defined in Table 2.

The symbols through which a data structure is encoded in computer memories are said to constitute a *storage structure*. For example, a rectangular matrix (a data structure) is usually represented as a string of storage words holding matrix element values that can be referenced through a base address and a displacement indexing scheme (a storage structure).

A *computer program* is the symbolic representation of a plan of computational activities that involve operations on data structures. These operations take memory-resident data structures as inputs (operands) and produce new data structures (results) that again reside in memories. The state behavior of the computing machine can be viewed as the time pattern of the data structures held in memories.

A computer program is realized as a set of machine instructions executed under the supervision of a *control structure* that specifies the order in which the operations are to be carried out. The machine component that executes a program is called an *instruction set processor*. Most machines have a unified instruction set processor for operating on arithmetic, logical and symbolic data-types; this hardware component is called the *central processing unit*, or CPU.

The set of machine instructions or instruction stream that constitutes a program may be viewed as nothing more than a specialized data structure. The physical representation of such instructions in a computer memory is called the *stored program*. (The ability for treating instructions and data in a similar fashion is a distinguishing feature of the modern digital computer). The set of programs that can be used on a particular computer system is called the *software*.

Data Descriptions

Description of data and of relationships between data objects are of two forms: *logical* and *physical*. Physical data descriptions refer to the manner in which the data is actually recorded in the storage hardware. Logical data descriptions refer to the manner in which the data appears to the application programmer or the user of the data. It is the function of the data management software to convert from the programmer's and and user's view of the data to the physical reality, and vice-versa. The linkage process is known as *binding* or *data mapping*.

Logical Data Descriptions

The logical data objects, i.e., data structures, that will be considered in this paper can be organized in a *logical hierarchy*, as depicted in Figure 1. Formal definitions of the data structures appearing in Figure 1 are given in Tables 2, 2A and 2B.

As strikingly illustrated by Table 3, the terminology pertaining to the logical data description is presently far from uniform. Much of the confusion stems from the fact that logical and physical descriptions of data structures were not clearly differentiated until the early 1970s. This explains why physical-description terms such as "field", "segment" and "file" still linger in the logical description of many data management systems. The term "record" is also a nonending source of confusion (some computer dictionaries list six or more meanings) but it has been left in Table 2, with the qualifier "logical", because of its widespread usage and the present lack of suitable alternatives.

It should also be noted that the categorization of the fundamental (but elusive) term *database* is far more detailed than that used in Part I.

Physical Description

Several terms that are frequently used in conjunction with the description of storage structures are collected in Table 4. With the exception of the multipurpose term *block*, this nomenclature is fairly well accepted. This reflects the fact that storage hardware technology is well established by now, whereas data management technology is still in a state of flux.

Table 1. Terminology Pertaining to Computer Storage Facilities (Arranged Alphabetically)

Term	Equivalent Term(s) and Abbreviations	Definition(s)
Address		See *storage address*
Auxiliary storage	Peripheral storage Secondary storage Backing store (Br.) External store (Br.)	Storage facilities of larger capacity and lower cost but slower access than main storage. Usually accessed via data channels (cf. *I/O device*), in which case data is stored and retrieved in physical-record blocks.
Bank		A grouping of homogeneous hardware elements motivated by access circuitry considerations.
Bit		An abbreviation of *bi*nary digi*t*. The term is extended to the actual representation of a binary digit in a storage medium through an encoded two-state device.
Byte		1. A generic term to indicate a measurable portion of consecutive binary digits (bits). 2. The smallest main storage unit addressable by CPU hardware. In machines with character addressing, byte and character (def. 2) are synonymous.
Character		1. A member of a set of elementary symbols that constitute an alphabet interpretable by the computer software. 2. A group of consecutive bits that is used to encode one of the above symbols.
Direct-access storage	*Random-access storage*	A type of storage wherein access to a position, at which data is to be stored or retrieved, is not dependent on the position at which data was previously stored or retrieved.
Extended core storage	Large core memory Bulk core Slow core ECS	A random-access, solid-state storage medium that shares many of the attributes of main storage, such as direct CPU addressing, but with larger access time and (often) block read/write constraints.
Facilities		See *storage facilities*
File		See *logical filename, physical file* (Table 4)
Input-output device	*I/O device*	An auxiliary storage device connected to the CPU by a data channel.
Logical device		See *storage device*
Logical filename		The identifier assigned to a (logical) storage device by the operating system.

Table 1. Terminology Pertaining to Computer Storage Facilities (Arranged Alphabetically) (Continued)

Term	Equivalent Term(s) and Abbreviations	Definition(s)
Main storage	*Primary storage* *High-speed memory* *Core* Internal store Main storage unit Small core memory	A fast, direct-access, electronic memory hardwired to the CPU. Holds machine instructions and data that can be accessed in a time of the order of a few machine cycles. Historically, the widest used form of main storage was magnetic core memory; hence the survival of the abbreviated designator "core".
Mass storage	Bulk storage	A generic term for online, large-capacity, direct-access auxiliary storage allocatable for public use. Generally implemented on rotating devices such as magnetic drums and disk packs.
Memory		See *storage*
Offline storage	Remote storage	Storage not directly controlled by the CPU.
Online storage		Storage under direct control of the CPU. It normally includes main storage, ECS (if any), mass storage, and active tape drives.
Page	Memory page Storage page	A unit of allocation of main and auxiliary storage in virtual memory computers and simulations thereof.
Permanent storage	Nonvolatile storage Catalogued storage	A type of auxiliary storage that retains information beyond the termination of the run-unit(s) that stored the information.
Random-access storage		See *direct access storage*
Sequential-access storage	Serial storage	A type of auxiliary storage (e.g., magnetic tape) in which data can only be accessed in the sequence in which it is stored.
Storage	*Memory* Store	Any device or medium that is capable of receiving information and retaining it over a period of time, and allows it to be retrieved and used when required.
Storage address	*Address*	1. A label, name or number that identifies the place where data is stored in a storage device. 2. The part of a machine instruction that specifies the location of an operand or the destination of a result.
Storage allocation		The process of assigning specific areas of storage to specific types of data.

Table 1. Terminology Pertaining to Computer Storage Facilities (Arranged Alphabetically) (Continued)

Term	Equivalent Term(s) and Abbreviations	Definition(s)
Storage device (logical)	*Logical file* Memory device Name space *Logical address space*	A subset of the storage space (def. 1) that is treated as a named entity by the operating system for purposes of allocating and releasing storage resources during the execution of a run-unit (task). The term is most often applied to auxiliary storage facilities.
Storage facilities		Hardware available to store data at a computer installation.
Storage pool	*Pool* Resource pool	A grouping of temporary storage resources for the common advantage of various processes or activities.
Storage space	*Data space* Information space Storage capacity Storage resources	1. The ensemble of storage facilities assigned to a run at a given moment. 2. The capacity of a storage device in terms of an appropriate storage capacity unit such as characters, words, or tracks.
Storage unit		A readily detachable part of the storage facilities (e.g., a magnetic tape drive).
Temporary storage	Volatile storage	Storage resources which are discarded during program execution or on run completion.
Virtual memory	Virtual storage	The simulation of large-capacity main storage by a multilevel relocation and paging mechanism implemented in the hardware.
Word	Main storage location Cell	The standard main-storage allocation unit for numeric data. A word consists of a predetermined number of bits, characters or bytes, which is addressed and transferred by the computer circuitry as an entity.

Note: Terms in italics are those commonly used in this paper.

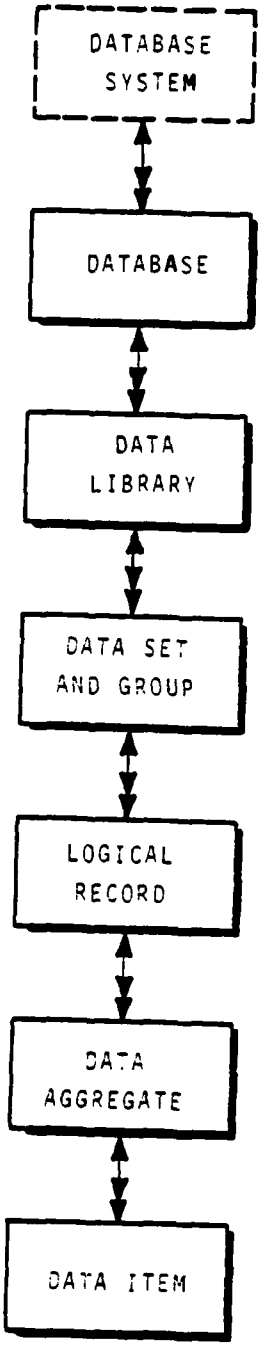


Figure 1. Logical Data Structure Hierarchy (single arrow indicates a one-to-one relationship, a double arrow a one-to-many relationship)

Table 2. Terminology Pertaining to Logical Data Objects (Arranged by Increasing Degree of Logical Complexity)

Term	Definition
Data item	The smallest quantum of information that can be referenced by name.
Data aggregate	A collection of data items, which is given a name and referred to as a whole. Five types of data aggregates deserving special mention are defined in Table 2A.
Logical record	A named collection of data items or data aggregates, which constitutes the basic transaction unit in a logical I/O operation.
Data set	A named collection of logical records.
Group	A data set containing a special "owner" or "master" record (the group directory) and a set of member records.
Data library	A named collection of data sets residing on permanent storage. It is the most complex data structure upon which a global database management system operates.
Database	In generic terms, a named collection of data organized according to a data model and serving a specific purpose. Further categorization of this term is given in Table 2B.
Database system	The set of all databases maintained on a computer installation (or computer network), which are administered by a common database manager.

Table 2A. Special Types of Data Aggregates

Term	Definition
Linear array	A data aggregate consisting of a set of homogeneous data items, called array elements, which are selected by a simple indexing scheme (i.e. the elements can be correlated with the sequence of positive integers.) If the data items are numeric, a linear array is often called a vector; if nonnumeric, a string.
Rectangular array	A data aggregate consisting of a set of homogeneous linear arrays. If the linear arrays are vectors, a rectangular array is sometimes called a rectangular matrix, or matrix for short. Each element of a rectangular array can be selected by a double indexing scheme.
Packet	A data aggregate consisting of a set of heterogeneous but logically related data items. (These are called *records* and *structures* in the PASCAL and C programming languages, respectively.)
Linked list	A data aggregate consisting of a string of homogeneous packets (sometimes called "atoms" in list-processing literature), which are linked by a pointer field mechanism.
Table	A data aggregate consisting of a string of symbol-packet pairs of the form (s,p), where s is a numeric or symbolic key, and p a packet containing a set of data items associated with s. If the packets reduce to linear arrays, a table is structurally equivalent to a rectangular array.

Table 2B. Further Categorization of the Term "Database"

Term	Definition
Distributed database	A global database physically distributed over several computer systems.
Global database (also *external database*)	A database residing on permanent storage and accessible by a network of communicating programs and users of such programs. In the hierarchical structure of Figure 1, a global database consists of one or several data libraries.
Help database	A global database containing program documentation text for online examination by interactive users of an integrated program network.
Local database	A database attached to, and accessible only by, an individual program.
Permanent database	In generic terms, a database residing on permanent storage devices. Specifically, the term applies to global databases as well as local database segments that may be reused by the generating program (e.g., restart data).
Project database	A global database whose data model responds to the needs of a project-type enterprise. It is generally organized according to a hierarchical model.
Temporary database (also *transient* or *volatile database*)	The portion of a local database that disappears upon run termination. Generally consists of auxiliary information, intermediate results and data structures copied to the global database.
Working database (also *running* or *operational database*)	The database upon which a running program operates at a given moment. It embodies its local database as well as possibly sections of the global database.

Table 3. Terminology Used to Describe Logical Data Structures in Various Data Management Systems

Source	Terminology[1]					
CODASYL[2]	Data item	Data aggregate	Record	Set	Database	
IBM[3]	Data element	Segment	Record	Data set	Data bank	
COBOL	Elementary item	Group item	Record	File		
IMS[4]	Field	Segment		File	Database	
MARK IV[5]	Field	Segment		File	Master file	
IDS[6]	Data field	Group item	Record	File	Master file	
TDMS[7]	Element	Repeating group	Entry	File	Database	
ASKA[8]		Paragraph	Page	Book	Net	Problem file
AID[9]			Record	Group	Section	
This paper	Data item	Data aggregate	Record	Data set, Group	Data library	Database

Notes:

1. Terms in the rightmost three columns are only approximately equivalent
2. Committee on Data System Languages, Data Base Task Group (ACM)
3. Informal use
4. IBM's Information Management System
5. Informatics, Inc.
6. Honeywell's Integrated Data Store
7. System Development Corp.'s Time-Sharing Data Management System
8. Schrem-Roy [2] and Schrem [3]; leftmost two levels available in current version.
9. Jensen [4].

Table 4. Terminology Pertaining to Physical Storage Structures (Roughly Arranged by Increasing Complexity)

Term	Equivalent Term(s) and Abbreviations	Definition(s)
Field	Elementary item Data field	The smallest unit of information storage that holds the value associated with some measure. If a field is broken down, it loses meaning.
Block	Data block X-block (X = word, core, tape, disk, etc.)	1. A generic term to denote a string of homogeneous storage objects such as characters, words, or fields, which are considered as a physical entity for some purpose. 2. A group of adjacent fields that is moved as a whole from one storage location to another, or from one storage medium to another. 3. A unit of media allocation that serves as a data transfer measure.
Sector		The smallest addressable unit in a rotating auxiliary storage device such as drum or disk. A sector normally consists of one or more words.
Physical record		A unit of auxiliary storage media that can be read or written without need to read and rewrite adjacent physical records. On rotating devices, a physical record consists of one or more sectors.
Extent	Segment Area	1. A collection of physical records, which are contiguous in auxiliary storage. 2. A contiguously addressed storage region. 3. The size of any such region.
Track	Band	1. The portion of a mechanical storage device such as drum, disk or tape, which is accessible to a given read/write station. 2. A unit of rotating auxiliary storage media characterized by most rapid transfer of consecutively recorded data.
Cylinder		For disk units with multiple read/write heads, all of the tracks that can be accessed without mechanical movement of the heads.
Volume	*Physical file* Tape reel Disk file Drum file	The storage space associated on a one-to-one basis with a logical storage device (cf. Table 1).

Section 3

PROGRAM ORGANIZATION

Centralized data management is most effective when used in conjunction with a highly-modular, structured program architecture. This section overviews concepts and practices deemed important in this regard. Readers interested in further details are advised to consult Yourdon and Constantine [5] and Yourdon [6]. These books, to which frequent references will be made, address structured software design and implementation, respectively. A glossary of terms often used in connection with program organization is provided in Table 5.

Logical Modules and Functional Hierarchies

Large-scale computer programs are undoubtedly an example of man's most complex artificial systems. There is a recursive feature to most system descriptions, and computer programs are no exception. The key element in a recursive description is the concept of *logical module*. There are various ways of defining a logical module, each of which reflects to some extent personal preferences as to program organization rules. For our purposes, the following informal definition should suffice:

> A logical module is a software element that performs a well-defined function.

A logical module should not be confused with its physical implementation in terms of machine instructions. A simple function might be implemented with a three-line code block or a reference to a system library routine. On the other hand, a complex computational task may demand the development of hundreds of subroutines. This distinction is elaborated upon in following subsections.

A measure of the complexity of a logical module may be obtained by introducing the concept of *module level*, which establishes a "most-basic-to-most-complex" hierarchy, much in the same spirit as the hierarchical classification of data structures (Figure 1). At the lowest level reside *atomic* or *primitive modules*, which perform the most elementary tasks worth distinguishing in a target description of the program. (They are primitive in the sense that it is known what they do as a whole, but the details of how it is done are irrelevant.) The next logical level is populated by modules that utilize primitive modules, and so on until the *primary logical level* is reached.

What is a primary logical module? If we are talking about an individual program, it is the program itself. If we are talking about an integrated program network (IPN), then a primary logical module is associated with what an IPN user would see as a primary activity. A clarifying example is provided in Section 4.

The preceding description assumes that a primary logical module can be described in a *bottom-up* fashion, i.e., proceeding from the simplest tasks to integrate more complex activity levels. There is an alternative description strategy known as the *top-down* approach, which is in many ways preferable. In this description primary tasks are identified first; then each primary task is broken down into secondary tasks, and so on until the target primitive level is reached. In the top-down design and implementation process the primitive level is gradually refined as the program structure takes shape; see, for instance, [5-7].

Physical Modules and Program Modularity

The term *program modularity* frequently evokes the restrictive notion of an entity built up of standardized, "plug-in" units. Although this model may be useful for building tightly-coupled, super-executive-bound IPNs, it is too restrictive for database-linked networks. A more relaxed definition of modularity, based on the physical module concept, will be admitted here.

> A *physical module* is a software element delimited by *closed* and *regular* interfaces.

An interface is called *closed* if the transmission of data is effected through address, stack or name pointers to the data. Two important variants are: the interface contains only call-by-reference arguments (a main-storage-closed interface) or names of database-resident data structures (a database-closed interface).

An interface is called *regular* if it protects the environment of the physical module from modifications in its source code or abnormal run terminations. The term *minimally coupled* is used in Ref. [5] in this regard. An interface can be regularized by enforcing common sense rules such as: (a) no in-place modifications of input database elements is allowed, (b) all major results are placed on the global database, (c) only one exit point is allowed, (d) standardized treatment of error or abnormal conditions, and the like.

Example 1. A Fortran subroutine using common blocks for communication with its calling program cannot be a physical module, because the interface is open. (This does not rule out, however, the use of common blocks to interface a *group* of subroutines that together constitute a physical module; in fact, this organization can be often recommended. What is essential is to forbid access to such blocks outside of the subroutine group.)

Example 2. A Fortran subroutine using nonstandard returns cannot be a physical module, because its interface linkage is not regular.

Example 3. A large-scale linear equation solver that replaces the global-database-resident input coefficient matrix with its factorization is not a physical module because it violates the interface regularity requirement. (Consider what could happen if the solver errors off.) The interface can be regularized, for instance, by copying the input matrix on a local database before executing the solver.

A physical module that possesses a global-database-closed regular interface and is executed as an independent program will be called a *primary physical module*, or *program module* for short.

A program will be said to be *modular* if its organization in terms of physical modules reflects the functional separation of its activities in terms of logical modules.

Self-Contained Modules and Utility Black Boxes

A *self-contained module* is a physical module that does not reference extraneous software, except perhaps standard system library routines. A self-contained module effectively embeds *all of its supporting software*, even at the cost of source code replication. Thus it can be used as a "plug-in" program-building element. Primitive physical modules such as a fast vector innerproduct procedure are the simplest examples. A data management system, a plot-generation library, an eigenvalue extraction package can be offered as examples of complex self-contained modules. A helpful way of visualizing a self-contained module during top-down program development is to think of the magic black box.

A *utility module*, or simply *utility*, is a self-contained module that performs a task of actual or potential interest to a class of programs. A collection of utility modules organized as a software package is a *program utility library* (which should not be confused with the term data library defined in Table 2). A good tutorial introduction to the use of primitive utilities as software-building tools is provided in Kernighan and Plauger [8].

Program Design Overview

Suppose that a programming team is faced with the construction of a database-linked IPN. If no *a priori* constraints enter into the picture, the following design strategy is recommended.

(D1) Identify primary logical modules.

(D2) Identify information exchanged by logical modules, and construct a data flow diagram.

(D3) Design problem-oriented data structures to carry the information.

(D4) Identify primary physical modules required to process the data structures (processors) and to direct processing activities (controllers).

The qualifier "problem-oriented" in (D3) means that the data structures must reflect the physical model. In other words, the *application drives the data structures*, and *data structures drive the program design*. The insistence upon using problem-related data structures is based on the structured-design observation that this practice leads to minimally-coupled program networks [5].

Example: in a finite element program, the concept of element is fundamental and all data structures should be built around it. Nodes (grid points), which are central in finite difference programs, play only a comparatively minor role. (In fact, it is possible to design and build finite element codes that do not use the concept of "node" at all.)

Network Implementation

The implementation of the IPN design is carried out by building the physical modules. In the database-linking approach, the use of existing software, and most especially existing utilities, has first priority. If it is necessary to develop a physical module from scratch, the following implementation steps are recommended.

(I1) First, write technical specifications and an users' manual draft for each physical module.

(I2) Prepare a test program (conversational if we are dealing with a primary physical module), and run it with stubs. Iterate with (I1).

(I3) Finally, code the stubs in top-down fashion.

(Note that this sequence: document-test-code, effectively reverses the usual bottom-up implementation sequence: code-test-document.)

It should be emphasized that these recommendations apply strictly to software products intended for a community of end users. They are not very relevant for special-purpose or research-oriented software, which is usually thrown away when done. These simpler programs rarely need any form of centralized data management.

Table 5. Terminology Pertaining to Program Architecture (Alphabetically Arranged)

Term	Equivalent Term(s)	Definition
Closed interface		An interface in which communication is effected by supplying addresses of data structures. Generally implemented by a formal calling sequence or by supplying names of database-resident structures.
Controller	Control module	A software element whose primary function is to direct the activities of other software elements.
Converter	Filter	A kernel processor with a single data structure input.
Integrated program network	Program network IPN	A set of controllers and processors (kernel or macroprocessors) communicating through a common *global* data manager.
Interface	Entry point set	The point (or set of points) in a software element at which control or data is received or transmitted.
Kernel processor		A processor characterized as (a) being self-contained, (b) generating only one data structure as main product.
Logical module	Transform element Bubble	The conceptual visualization of a software element as a "data machine" that performs a specific function.
Macroprocessor		An assembly of kernel processors communicating through a common *local* data manager.
Manager (Data)		A software element whose primary function is to store, maintain and retrieve data.
Physical module	Module Box	The implementation of a logical module as a cohesive software element with closed and regular interfaces.
Primary physical module	Program module	A physical module implemented as an independently executable program.
Processor	Computational module	A software element whose primary function is the production of data structures.
Regular interface		An interface organized in such a way that the software element is *minimally coupled* with its environment, in the sense of [5].
Self-contained module	Software kernel Black box	A physical module that does not reference external software (other than perhaps system-library routines).
Software element	Software unit Program component	A bounded aggregate of computer-processable statements identified by a name.

Table 5. Terminology Pertaining to Program Architecture (Alphabetically Arranged) (Continued)

Term	Equivalent Term(s)	Definition
Utility	Software tool	A self-contained module that provides actual or potential support to several higher-level modules.

Section 4

DATA MANAGEMENT SYSTEMS

Definition

A data management system (DMS) is a self-contained utility module that centralizes activities pertaining to the management of data structures. The main duties of a scientific data management system are:

Resource Allocation. Assignation of resources to hold storage structures representing the logical data structures managed by the DMS.

Storage and Retrieval. The transfer of a storage structure, or sections thereof, from one recording medium to another within a hierarchical memory environment.

Cataloguing and Bookkeeping. Maintaining an access directory of all data structures under DMS control.

When the set of data structures under control of the DMS constitutes a database (i.e., is endowed with an appropriate data model), the DMS is called *a database management system* (DBMS) or *database manager* (DBM) for short.

The marriage of a DMS to an applications program results in the appearance of three elements in the data management game. They are:

User Program. This term denotes all applications program software external to the DMS. The user program *drives* the data manager in a master-slave configuration.

Data Manager. Inasmuch as the DMS is a self-contained module, the slave appears to the master as a black box with multiple entry points.

Data Space. This term (defined in Table 1) denotes the union of data structures pertaining to the user-program/DMS complex, as well as the representation of this data in the form of storage structures. (Qualifiers "logical" and "physical" may be used to distinguish between the two data description viewpoints as necessary.)

The interaction of user program and data manager can be succintly described in twenty words: *The data manager is the carrier and depositary of information; the user program is the producer and consumer of information.*

Segregation of the Data Space

When a scientific DMS is running in conjunction with a user program, a three-way segregation of the data space occurs (cf. Figure 2).

User Program's Own Data. The portion of the data space that remains under exclusive control of the user program.

Working Database. The portion of the data space that is placed under "joint administration" of the user program and the data manager. The latter is responsible for state management tasks at or above the logical record level. The user program is responsible for creating and making use of the information contained within logical records, i.e., activities taking place at the data item and data aggregate levels. (A further segregation of this space occurs when the distinction between global and local database is introduced, as discussed below.)

Manager's Own Data. Data maintained by the DMS to effect its own administration and to keep track of storage structures residing on the working database. This data is generally inaccessible to the user program except for state reporting (display) operations.

Global and Local Databases

A clear-cut separation between global and local databases was proposed in Part I as a promising way of alleviating the problem of physical data dependence. As a bridge to implementation details of Part III, we expound further on the functional characterization of this concept.

The global database (or, more precisely, its data model) is what the human user perceives. Global databases hold not only application data, but operational data such as command language procedures and help files, which an interactive user can access on a system-wide basis. The key attribute is *ease of use.* To accomplish this, human engineering concepts must enter into the design, i.e., the data must be highly structured and profusely self-described. Inasmuch as database libraries have to survive individual runs, they must reside on permanent storage devices. (Unfortunately, the latter are notoriously unsuitable for efficient online processing.)

The local database is what a running program uses. The key attribute here is *resource utilization efficiency.* To accomplish this, local databases can be conveniently implemented within a virtual storage organization. This organization can be either hardware-implemented (in virtual memory machines) or software-simulated.

To further fix the ideas, consider a voluminous data structure such as the master stiffness matrix of a finite element model. On a local database such a matrix might be physically distributed over hundreds or thousands of *pages* spread out over main storage, extended core, paging drums or fixed-head disk files. If saved on the global database, the stiffness matrix would typically reside on a slow-access permanent file device as a sequence of logical records. The latter configuration simplifies archival on serial access devices such as tapes or cassettes, and also facilitates file transmission among computers (in the case of a distributed global database).

The configuration of local databases can be expected to be highly volatile as the developer modifies, augments, or "tunes up" processors. On the other hand, global data structures must be subject to strict representation and self-description standards to insure network operational stability. Here we perceive another role of the global database: to act as a "buffer" that protects users from day-to-day software modifications.

An Illustrative Example

Conceptual block diagrams such as Figure 2 (or Figures 3 and 4 in Part I), tend to conceal the complexity of real-life applications. To give a more realistic picture the example of a database-linked, interactive-oriented, preliminary design system for large space structures is offered as an example. Figure 3 is a data flow diagram that depicts primary logical modules and major pieces of information exchanged among those modules. Most of the structured-chart conventions of Ref. [5] have been followed in drawing this diagram. In particular, logical modules are shown as "bubbles"; while database structures flowing among bubbles are identified by circle-tailed arrows. (Only some of the latter have been described with annotations so as not to clutter the picture.) As things presently go, this IPN can be categorized as being of intermediate complexity.

Now Figure 3 is a "flat" functional chart, which is suitable for explaining the design process in general terms, but does not properly portray the *implementation* of the IPN architecture. A three-dimensional representation, such as the one sketched in Figure 4, is more appropriate. Three "operational planes" can be distinguished: (U), the end user's brain; (P), the set of primary physical modules, or program base; and (D), the global database (local database planes are omitted for clarity). Two software elements are used to bind these planes. The global database manager binds (D) and (P). The program control system (PCS) binds (U) and (P).

Program Control System

Although not treated in this paper, we note that the PCS is simply the implementation of a program base model, i.e., the "lens" through which the end user perceives the physical modules. In an advanced implementation, the program base is conveniently organized as a relational model, and manipulated through an appropriate command language based on relational calculus or algebra [9]. We hasten to point out that there is no relation whatsoever between a PCS and what is called a super-executive in Part I. The PCS is only a "consulting module" that suggests how to run the programs to accomplish certain objectives, but the user remains in control of final decisions.

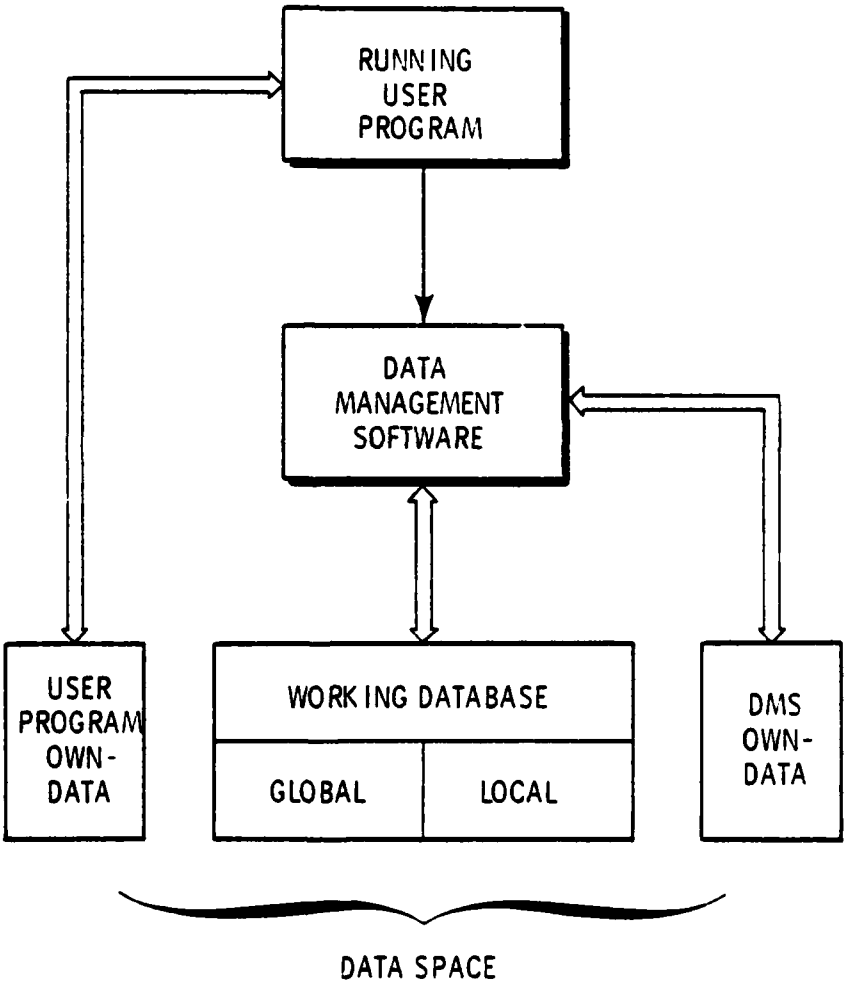


Figure 2. Operational Components of a DMS-Linked Running Program

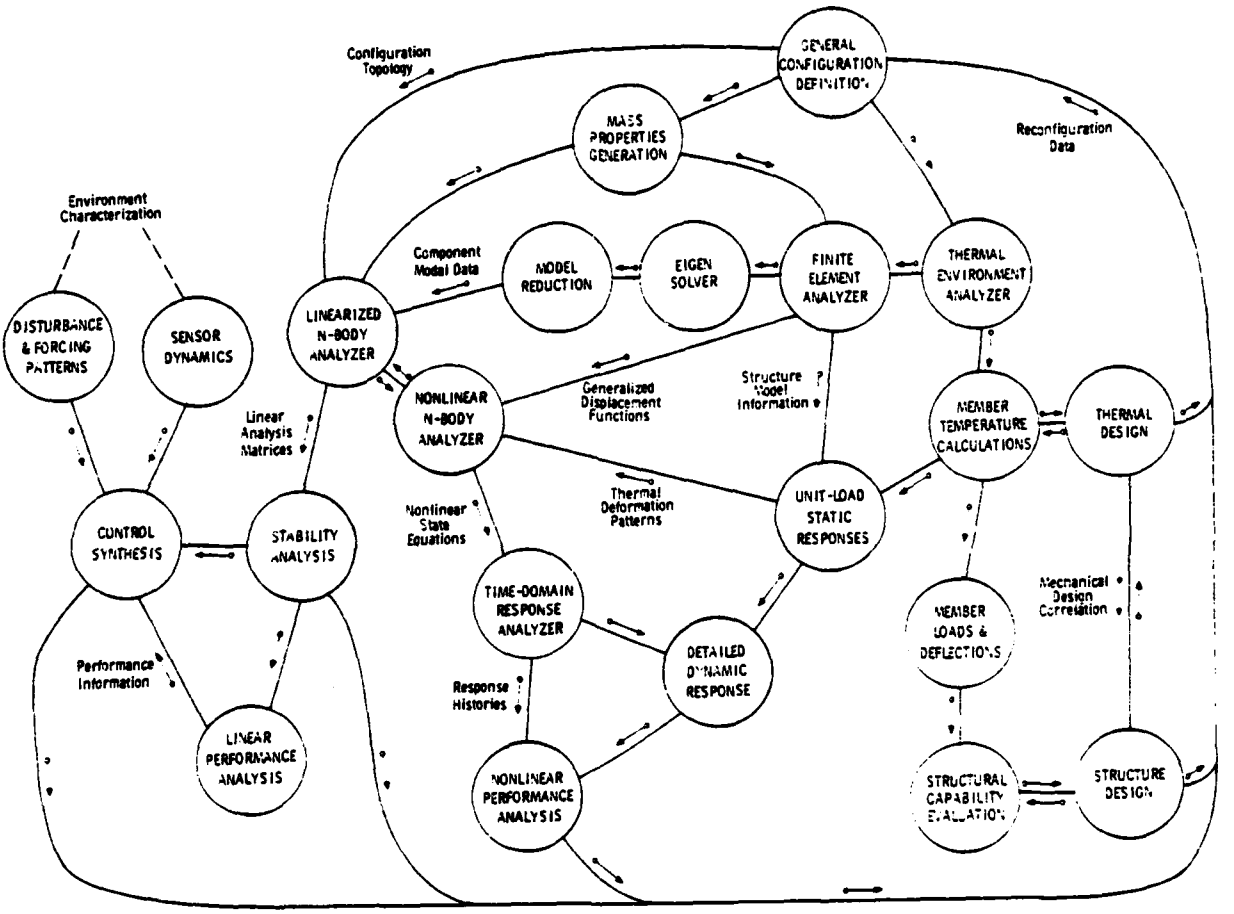


Figure 3. High-Level Data Flow Diagram of IPN Example: Interactive Preliminary Design of Large Space Structures

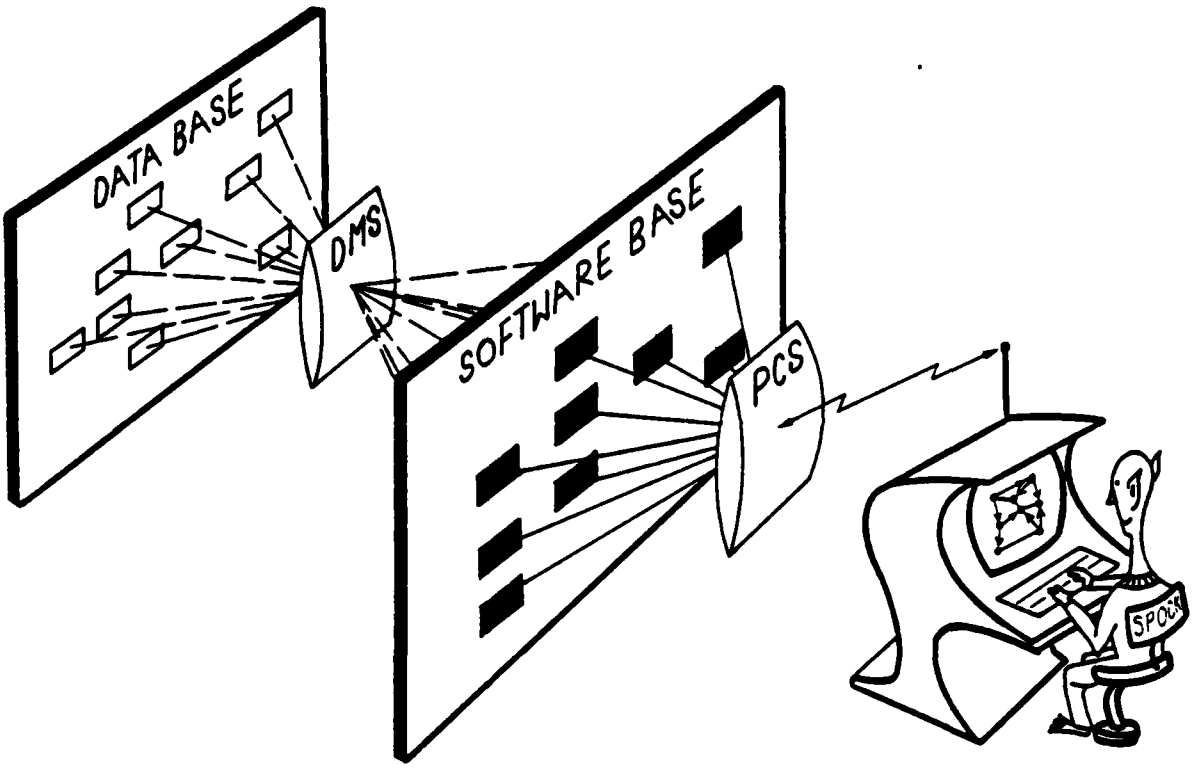


Figure 4. Three-Dimensional Visualization of the Operation of the IPN Example

Section 5

ACTIVITY LEVELS

The Concept of Layered Data Management

A data management system is seldom written to be all things to all programs running on a computer installation, as are some large-scale operating systems. Performance considerations dictate otherwise. A DMS should be limited in its domain of applications to classes of programs sharing similar operational characteristics. Examples of such classes are: business processing programs, scientific computation programs, real time software and language processors. The DMS limitations are reflected in the following areas.

1. Types of data structures accepted by the DMS;

2. Procedures for representing such structures and accompanying relationships in computer memories; and

3. Spectrum of operations allowed on the storage structures.

The restriction of a DMS to serve certain classes of user programs fits within the approach called "design according to performance specifications" as opposed to "design according to functional specifications". The former approach emphasizes realistic, clearly defined goals, and uses the simplest techniques to achieve them. An instance of this approach is the selection of *hierarchical* data structures for data management. Expanding goals to encompass a wider range of applications and services risks overcomplexity and inefficiency, to say nothing of the reaction of application program developers forced to learn to use and deal with a gargantuan system.

The operational distinction between business and scientific programming is one of the main themes of this paper. It would be a gross error, however, to regard all scientific programs as equivalent; in fact, many "throwaway" scientific programs oriented to feasibility studies do not need a DMS at all! A compromise that permits a scientific DMS to provide a useful range of capabilities while minimizing performance degradation is the concept of "layered" or multilevel DMS.

A layered DMS possesses several hierarchically self-contained *activity levels*. Each level is implemented as a program module that makes no use of higher levels. The using program developer selects the level according to estimated requirements. As the level increases, so do the technical capabilities of the DMS and the complexity of the data structures managed. This increase in functional capabilities is paid in terms of storage overhead and, in the case of an existing program being DMS-linked, in the need for more or less extensive rewriting of the original source code.

To specifically illustrate the concept of layered DMS, Figure 5 shows the general configuration of the system described in Part III. Five activity levels can be distinguished: I/O manager, direct-access library manager, virtual memory manager, resource pool manager, and global database manager. A functional description of these components is not necessary at this point. Figure 6 shows how the various levels can be used in support of an user program. We note that in practice it is quite common to start with a very simple configuration such as the one shown in Figure 6(a), and gradually "move to the top". This process is facilitated by the fact that the activity levels are self-contained modules.

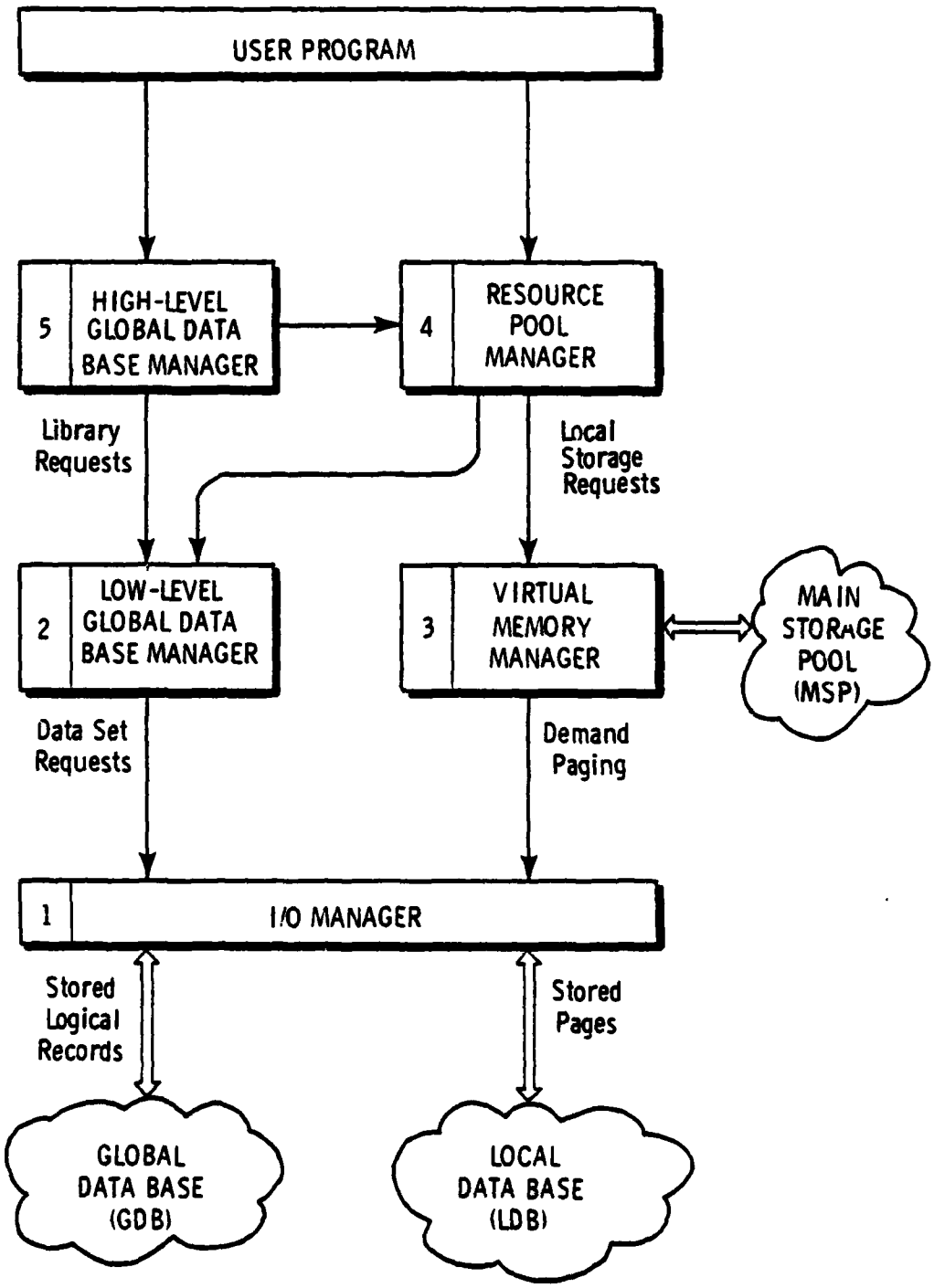


Figure 5. A Layered (Multilevel) Data Management System

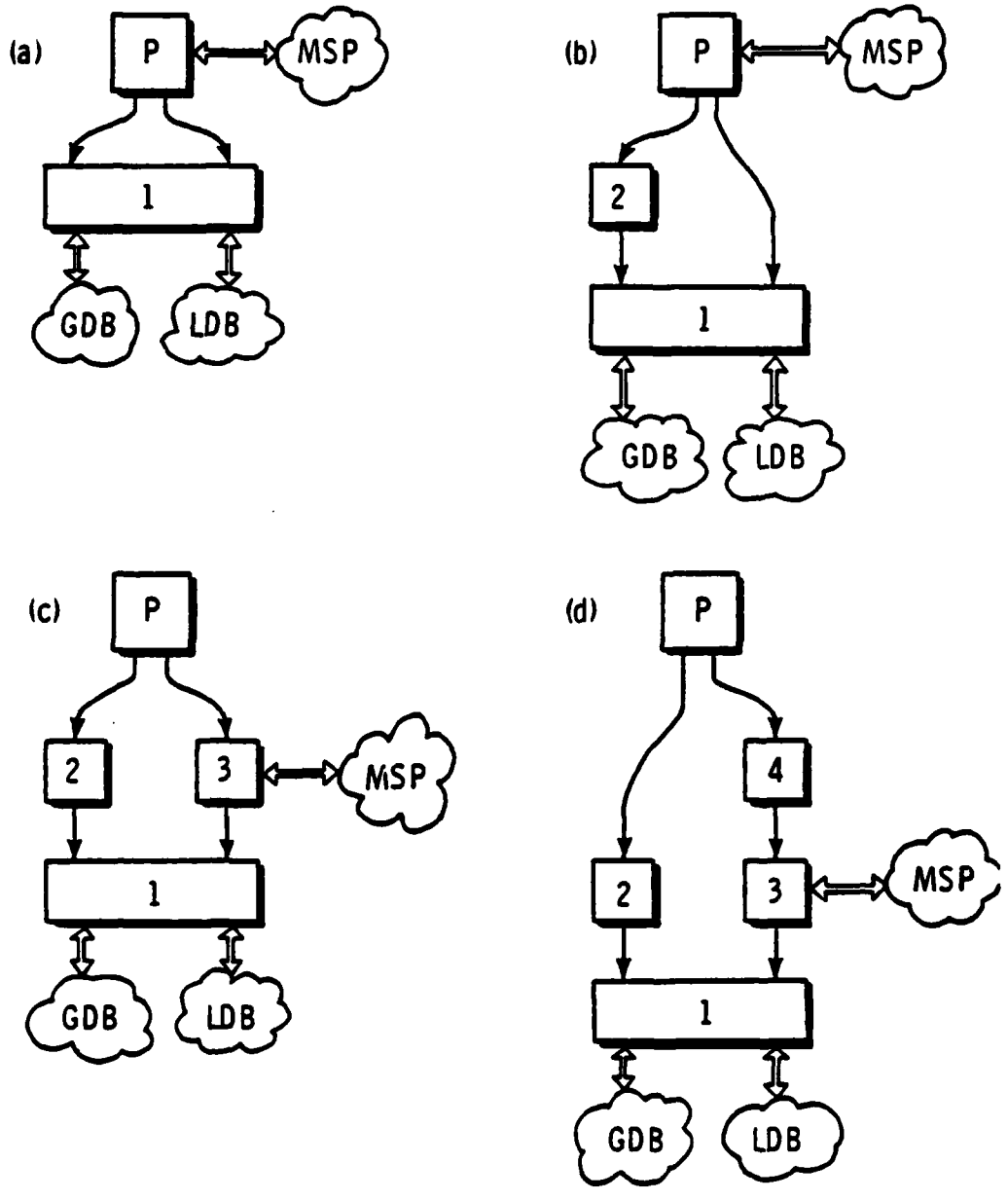


Figure 6. Sample Configurations in which the Layered Data Management System of Figure 5 Can Be Used in Support of a User Program (P)

Application to Program Networks

The possibility of adapting DMS configurations to user program characteristics is particularly important for integrated program networks. To illustrate this point, consider the following opposite situations.

Satellite Processors. Examples: input data preprocessors, result postprocessors, data converters. Main features: complex and unpredictable interactions with global database; small volume of internal computations; local resource management unimportant; close rapport with interactive users. Conclusion: the local data manager may be very simple (or even entirely missing leaving the global manager in total control).

Number-crunching Processors. Examples: large-scale equation solvers, eigensolvers, time integrators. Main features: complex, huge local database; efficient local resource management imperative; relatively simple interactions with global database. Prescription: sophisticated local data manager, low-level global data manager.

Macro and Micro Data Management

The basic transaction unit of a scientific DMS such as the one depicted in Figure 5 is the logical record. The DMS does not care about the contents of the records; their interpretation is reserved to the user program. As noted in Part I, the DMS operates primarily as a "filing system". This viewpoint is certainly adequate for computational modules as well as pre- and post-processors modules of application programs as used today.

As scientific programs keep growing in logical complexity and interact further with design and manufacturing processes, some modules will eventually acquire characteristics typical of information retrieval systems. Consider, for example, the following two cases.

Result condensation. Results of a series of engineering analyses are stored in a data library. The user wants to access the results, identify critical response conditions, and move them to a result-summary library.

High-level analysis setup. The analyst must perform a transient nonlinear analysis of a fluid-structure problem. He queries (through a program control system such as the one depicted in Figure 4) a program base for modules capable of carrying out this task. Then he requests that the PCS fabricate (put together) an assemblage of such modules. The result is a skeleton control card runstream which can be examined interactively and eventually submitted to the operating system.

To respond to these type of requests, it is natural to assemble utility modules with the external appearance of data managers. But these "managers" have necessarily to look inside record contents; in other words, to operate at the data item or data aggregate level. We call them *micro data managers*, or *micromanagers* for short.

Micromanagers can display many of the attributes of business data management systems, and are most conveniently organized in terms of *relational data models*. This is in contrast to the (macro) managers discussed so far, which work nicely with hierarchical models such as the one depicted in Figure 1. The main motivation for relational models lies in the fact that they provide high flexibility in query-oriented systems while still allowing for a modular implementation. All data structures handled by such models appear in the form of tables (defined in Table 2A), which represent n-ary relations among data items.

The concepts just discussed are illustrated in Figure 7. Note that the (macro) data manager appears to the micromanager as the *access method* or *data retrieval system* with respect to the hierarchically-organized data structures stored in the global and local databases. Configurations of this type will probably emerge in advanced scientific programs during the 1980s. Barring the unexpected, this process ought to culminate in the introduction of artificial intelligence levels (a level in which the query controller writes programs or modifies its own logic) in the late 1980s or early 1990s. Pioneering efforts along these lines have been recently discussed by Melosh *et.al.* [10] and Fenves [11].

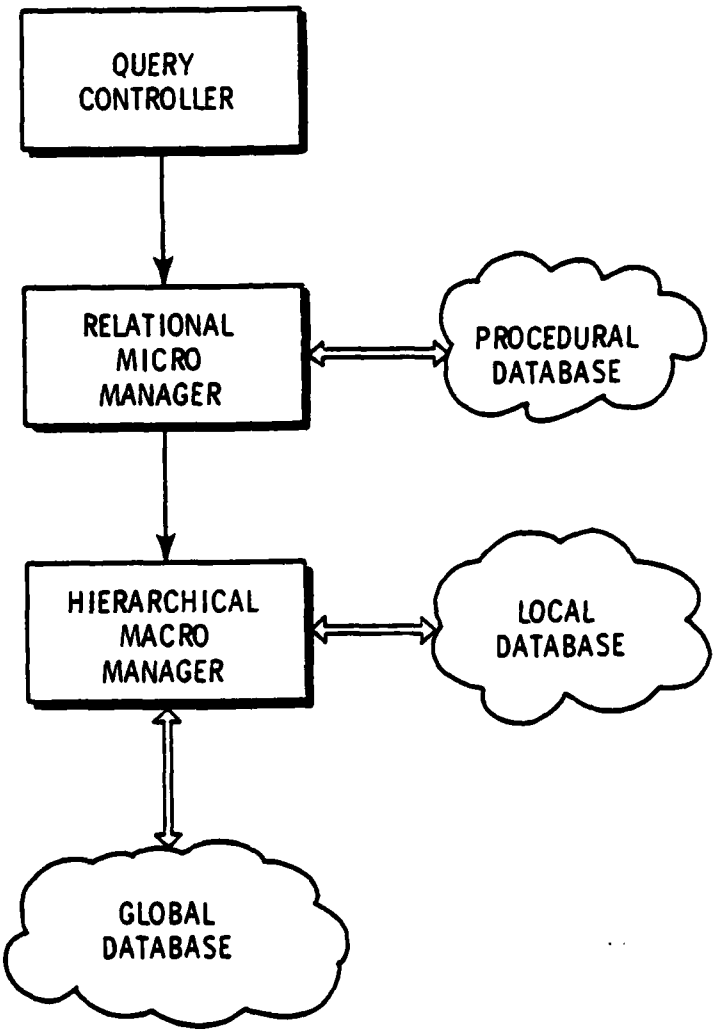


Figure 7. Macro- and Micromanagement Illustrated in Conjunction with the Use of a Query Controller

REFERENCES

1. C. A. Felippa, Database Management in Scientific Computing, I. General Description, *Computers and Structures*, Vol. 10, Nos. 1/2 pp. 53-61 (1979)

2. E. Schrem and J. R. Roy, An Automatic System for Kinematic Analysis, in *High Speed Computing of Elastic Structures* (Edited by B. Fraejis de Veubeke), University of Liege, Belgium, pp. 477-507 (1971)

3. E. Schrem, Computer Implementation of the Finite Element Procedure, in *Numerical and Computer Methods in Structural Mechanics* (Edited by S. J. Fenves, N. Perrone, A. R. Robinson and W. C. Schnobrich), Academic Press, New York, pp. 79-122 (1971)

4. P. S. Jensen, Storage Management for Numerical Processes, presented at Mathematical Software II Conference, Purdue University (1975)

5. E. Yourdon and L. L. Constantine, *Structured Design*, Prentice-Hall, Englewood Cliffs, New Jersey (1979)

6. E. Yourdon, *Techniques of Program Structure and Design*, Prentice-Hall, Englewood Cliffs, New Jersey (1975)

7. R. C. Tausworthe, *Standardized Development of Computer Software*, Prentice-Hall, Englewood Cliffs, New Jersey (1977)

8. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Reading, Mass. (1976)

9. C. J. Date, *An Introduction to Database Systems*, 2nd edition, Addison-Wesley, Reading, Mass. (1978)

10. R. J. Melosh, L. Berke and P. V. Marcal, Knowledge-Based Consultation for Selecting a Structural Analysis Strategy, in *New Techniques in Structural Analysis by Computer* (Compiled by R. J. Melosh and M. Salama), American Society of Civil Engineers Preprint 3601, ASCE Convention & Exposition at Boston, Mass., (1979)

11. S. J. Fenves, Potential Application of Artificial Intelligence for Evaluation of Analysis Results, in Preprint Vol. quoted in Ref. 10

12. C. Gordon Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York (1971)

13. S. Rosen, Electronic Computers: A Historical Survey, *Computing Surveys*, Vol. 1, No. 4, pp. 197-212 (1969)

14. E. L. Wilson, *SMIS, Symbolic Matrix Interpretive System*, University of California, Berkeley, Department of Civil Engineering, Report UCSESM 73-3 (1963)

15. O. L. Anderson, SNARK (TEL 237), Boeing Document D6-29769TN, The Boeing Co., Renton, Wa. (1969); see also: R. L. Dreisbach and G. L. Giles, the ATLAS Integrated Structural Analysis and Design Software System, in *Research in Computerized Structural Analysis and Synthesis - Research in Progress Papers* (Compiled by H. G. McComb, Jr.), NASA Conf. Pub. 2059, NASA Langley Research Center, Hampton, Va. (1978)

16. D. S. Warren, Applications Experience with the FORMAT Computer Program, *Proc. II Conference on Matrix Methods in Structural Mechanics*, AFFDL-TR-68-150, Air Force Flight Dynamics Laboratory, Wright-Patterson AFB, Dayton, Ohio, pp. 839-868 (1968)

17. G. J. Wennagel, H. H. Loshigian and J. D. Rosenbaum, RAVES - Rapid Aerospace Vehicle Evaluation System, in *Integrated Design and Analysis of Aerospace Structures* (Edited by R. F. Hartung), Winter Annual Meeting of The American Society of Mechanical Engineers - Houston, Texas, Nov. 1975, ASME, New York, pp. 23-56 (1975)

18. E. L. Wilson, CAL - A Computer Analysis Language for Teaching Structural Analysis, in *Trends in Computerized Structural Analysis and Synthesis* (Edited by A. K. Noor and H. G. McComb, Jr.), Pergamon Press, Oxford, pp. 127-132 (1978)

19. R. H. MacNeal (ed.), *The NASTRAN Theoretical Manual*, NASA SP-221 (1969)

20. F. J. Douglas (ed.), *The NASTRAN Programmers Manual, Parts I and II*, NASA SP-223 (1969)

21. W. D. Whetstone, *SPAR Structural Analysis System Reference Manual*, NASA-CR 145098, Engineering Information Systems, Inc., San Jose, Cal. (1977)

22. W. D. Whetstone, Engineering Data Management and Structure of Program Functions, in Preprint Vol. quoted in Ref. 10

23. H. A. Kamel and M. W. McCabe, *GIFTS System Manual*, Dept. of Aerospace and Mechanical Engineering, University of Arizona, Tucson, Arizona (1978)

24. R. Damrath and P. J. Pahl, Data and Storage Structures for a System of Finite Element Programs, in *Formulations and Computational Algorithms in Finite Element Analysis* (Edited by K. J. Bathe, J. T. Oden and W. Wunderlich), MIT Press, Cambridge, Massachussets, pp. 140-162 (1977)

25. L. A. Lopez, FINITE: An Approach to Structural Mechanics Systems, *Int. J. Numer. Meth. Engrg.*, Vol. 11, No. 5 (1977)

26. L. A. Lopez, R. H. Dodds, D. R. Rehak, and J. Urzua, Data Management Applied to Structural Problems, *Proc. ASCE Conference on Computers in Civil Engineering*, Atlanta, Ga. (1978)

27. Feasibility Study of an Integrated Program for Aerospace Vehicle Design (IPAD), NASA CR 132390-97, The Boeing Co., Seattle, Wa. (1973)

ACKNOWLEDGEMENT

The preparation of this paper has been supported by the Independent Research Program of Lockheed Missiles & Space Co., Inc.. Ms. Colleen Miller drew the illustrations.

Appendix A

DATA MANAGEMENT IN COMPUTERIZED STRUCTURAL ANALYSIS

Computerized structural analysis, especially in the aerospace industry, has traditionally led the way in scientific data processing. It is sufficient to mention some of the advances that can be credited to this field:

Finite element methods (*ca.* 1950)

Direct assembly of discrete governing equations (*ca.* 1956)

Large-capacity direct equation solvers (*ca.* 1960)

Generalized array processing libraries (*ca.* 1964)

Integrated program networks for engineering design (*ca.* 1968)

Distributed engineering analysis (*ca.* 1975)

In briefly reviewing the evolution of centralized data management in scientific computing, it is therefore sufficient to look at structural analysis programs and integrated networks centered about such programs.

Major Accomplishments

Table A.1 correlates major accomplishments in computerized structural analysis technology (left two columns) against representative advances in general computing technology (hardware & software tools) and centralized data management.

We call attention to the existence of various time lags, three to be exact.

1. Lag between formulation advances and implementation: 0-2 years for special-purpose codes, 4-6 years for general-purpose codes.

2. Lag between advances in computer technology and effective utilization of these advances: 2-5 years.

3. Lag between advances in data management technology and effective utilization in scientific programs: 5-10 years.

Reasons explaining the larger third gap are offered in Part I. We now proceed to review the application of data management technology over the past 15 years or so.

Computing Environment Evolution

It is safe to say that no attempts at generalized data management, or even resource management, were made prior to 1965. Most scientific data processing systems used before this date were uniprogrammed machines with magnetic tapes as standard auxiliary storage (and archival) medium. As these systems could only process one job at a time, there was *no* incentive in trying to optimize the use of resources other than CPU time. In fact, a very common practice was to configure production-level structural analyzers in such a way that the entire high-speed (core) memory was used up, regardless of whether the problem required it or not. The largest processable problem was then defined by limitations imposed by the static dimension of Fortran arrays.

To increase processable problem size, very elaborate link and overlay schemes begin to be used in the early 1960s to "shoehorn" program segments into available high-speed memory. This practice has unfortunately survived until the present days (programmers' habits change more slowly than computing technology). To a large extent, this philosophy can be considered responsible for highly overlaid scientific program "monsters" growing out of control of the developers.

Multiprogramming was pioneered by Burroughs and Control Data in the early 1960s. By the late 1960s, it had become standard on most conventional mainframes (with the exception of IBM, which did not join until the announcement of the 370 series). The concurrent appearance of time sharing and minicomputers contributed to a gradual change in the computing environment. Although paged virtual memory appeared also in the early 1960s (on the ATLAS computer and later on Burroughs'), it does not seem to have significantly influenced the mainstream of scientific data processing.

By the early 1970s, the need to adapt the program to the problem had been recognized. That is, resources used should be commensurate with problem size. The popularization of user-allocatable, direct-access mass storage and (later) the availability of a hierarchy of main storage devices widened the range of resources that had to be managed.

For excellent accounts of the pre-1970 period in computer technology, see Bell & Newell [12] or Rosen [13].

Resource Management in Matrix Structural Analysis

Early attempts at generalized resource management centered upon dynamic array allocation. Simple core management schemes appeared in various *matrix oriented* codes written in the period 1962-1970. Typical of these were SMIS [14] in the university environment, SNARK [15], FORMAC [16] and RAVES [17] in the industrial environment. The SMIS program, now superseded by CAL [18], was especially important in that it set standards for computer-assisted instruction in matrix methods for over a decade, and branched out into innumerable batch and time-sharing versions.

NASTRAN [19, 20] and ASKA [2, 3] can be viewed as representative of the large-capacity, general-purpose, executive-linked codes that evolved in the period 1965-1970. In both cases, resource data management and matrix processing logic were entwined by today's standards. It must be remembered, however, that finite element structural analysis was often labelled "matrix methods" at the time; i.e., the appearance (direct algebraic formulation) was mistaken by the essence (approximation theory).

The architecture of NASTRAN was fairly advanced for the time during which design specifications were written (early 1960s). The concept of super executive was carried out to extremes. The NASTRAN executive in fact replaced the operating system and embodied both input/output supervisors and program linkage editors. This drastic measure was required because the operating systems of the time failed to meet ambitious design specifications such as management of online devices and multilevel program segmentation.

The fundamental NASTRAN data structure was the "matrix column record". This unhappy decision, in retrospect a spinoff of the Fortran column-by-column array storage and tape-like I/O mentality, succeeded in tying up the program into knots. It is an example of how confusion of logical and physical data structures can result in expensive mistakes. (In all fairness, the distinction was not well understood at the time.)

The ASKA data management system was firmly tied up to the "hypermatrix" concept. This is probably an outgrowth of the drum-paged UNIVAC machine on which the first version evolved in the late 1950s. The page-oriented, multilevel treatment of the data displays, however, far more logical flexibility than NASTRAN. In the "data retrieval system" described in [3], a three-level tree structure appears (expanded to five in later versions). The distinction between logical and physical data descriptions facilitated adaption to different problem-solving and machine environments.

Recent Trends and Developments

Modern database management (by which is meant emphasis on logical and conceptual data descriptions) timidly appeared during the early 1970s on a handful of structure analyzers. Although steady progress has been recorded since, the general technology level is far from that achieved in business data processing. Two main trends, related to the "degree of closeness" deemed desirable between data management and applications, can be distinguished.

Applications-Independent Data Management. The DMS is designed as a self-contained, separable utility logically divorced from the meaning of the data it manages. This *context-free* viewpoint is stressed in this paper. It is reinforced by current trends in structured programming (exact tools for each level, functional cohesiveness, minimal context coupling), but is tempered by the performance considerations mentioned in Section 5.

Applications-Dependent Data Management. The DMS, although centralized, is "made to order" to support a specific program, or program network. The manager operates on the basis of the *context* of the data it sees, and cannot be easily separated from the applications software.

How can these types be recognized? A simple "patch test" (apologies to Bruce Irons) can be applied. Try to visualize the DMS in question as being linked to support software of progressively more foreign nature: another structural analyzer, a finite-difference hydrodynamics code, a preliminary design package, a manufacturing control system, a batch payroll program, a real-time airline reservations system. Where can one draw the line? The further the "patch" works, the more context-free the manager is. (Of course, a positive answer does *not* imply that such a liaison would be ever desirable, as performance considerations intervene.)

In practice pure archetypes are seldom observed while hybrids abound. Moreover, in the case of a multilevel data manager it is common to see highly context-free low-level components serving application-oriented higher levels. An advanced example of this (intelligent analysis controller) has been depicted in Figure 7.

Representative examples of the context-free organization are the AID manager [4], the data-complex managers of the SPAR network [21] and its successor EAL [22]. Representative examples of the highly applications-coupled case are GIFTS [23] and ITS [24]. The POLO-FINITE system [25, 26] and the under-development IPAD system [27] can be offered as examples of a middle-of-the-road approach.

The functional distinction between local and global databases is a post-1975 concept conceived by the author while thinking about the lessons of Watergate (i.e., the two-tier image perception problem). Further elaborations of the theme of disjoint global databases can be expected in the near future as distributed and personal computing finally engulfs computerized engineering analysis.

Table A-1. Evolution of Computerized Structural Analysis

Year	Structural Analysis Technology		Computer Technology	
	Formulation	Implementation	General	Data Management
1950	Matrix formulation Force method	Equations set up by analyst	UNIVAC I Magnetic tapes Paging drums	
		Matrix subroutines		
	Direct stiffness method	Programmed equation generation	Fortran I IBM 704	
	Incremental nonlinear analysis	High-order languages	Monitor systems I/O channels IBM 7090	
		Block solvers		
		Iterative solvers		
1960	Modal dynamics	Substructuring	Disk files Virtual memory	
	Variational formulation		Operating systems	
		Band solvers	Multiprogramming	
	Refined elements		Fortran IV	
	Direct time integration dynamics		CDC 6600	Resource management
	Iterative nonlinear analysis	General purpose linear analyzers	IBM 360	
			Time sharing	
	Isoparametrics	Problem-oriented languages	Interactive graphics	Hierarchical databases
	Reduced integration			
1970		Profile solvers	Permanent disk-resident files	Network databases DBTG Report
	Mathematical foundations	Wavefront solvers General purpose nonlinear analyzers	Minicomputers	Relational databases
	Boundary elements	Program networks	Large scale integration	
		Database-linked IPN	Microcomputers	
			Distributed processing	Database machines


DAT
FILM